UNIT 3

TUPLES AND DICTIONARIES

3.1. Tuples

A tuple is an immutable sequence of Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

3.1.1. Creating a Tuple

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also.

Example:
```
tup1 = ('physics', 'chemistry', 1997, 2000);
tup2 = (1, 2, 3, 4, 5 );
tup3 = "a", "b", "c", "d";
```

The empty tuple is written as two parentheses containing nothing

Example:
```
tup1 = ();
```

To write a tuple containing a single value you have to include a comma, even though there is only one value.

Example:
```
tup1 = (50,);
```

3.1.2. Accessing values in a Tuple

There are various ways in which we can access the elements of a tuple.

1. Indexing:
   We can use the index operator [ ] to access an item in a tuple, where the index starts from 0. The index must be an integer, so we cannot use float or other types. This will result in TypeError. Trying to access an index outside of the tuple index range will raise an IndexError.

   Example:
   ```
   my_tuple = ('p','e','r','m','i','t')
   print(my_tuple[0])
   print(my_tuple[5])
   ```

   Output:
   ```
   p
   t
   ```

2. Negative Indexing:

   Python allows negative indexing for its sequences. The index of -1 refers to the last item, -2 to the second last item and so on.

   Example:
   ```
   my_tuple = ('p', 'e', 'r', 'm', 'i', 't')
   print(my_tuple[-1])
   print(my_tuple[-6])
   ```

   Output:

   ```
   t
   p
   ```

3. Slicing:
   We can access a range of items in a tuple by using the slicing operator colon (:).

   Example:
   ```
   my_tuple = ('p','r','o','g','r','a','m')
   print(my_tuple[1:4])
   print(my_tuple[:-5])
   print(my_tuple[5:])
   print(my_tuple[:])
   ```

   Output:
   ```
   ('r', 'o', 'g')
   ('p', 'r')
   ('a', 'm')
   ('p', 'r', 'o', 'g', 'r', 'a', 'm')
   ```

3.1.3. Updating Tuple

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples.

Example:

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');
# Following action is not valid for tuples
# tup1[0] = 100;
# So let's create a new tuple as follows
tup3 = tup1 + tup2;
print tup3;
```

Output:
```
(12, 34.56, 'abc', 'xyz')
```

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded. To explicitly remove an entire tuple, just use the **del** statement.

Example:

```
tup = ('physics', 'chemistry', 1997, 2000);
print tup;
del tup;
print "After deleting tup : ";
print tup;
```

Output:

```
('physics', 'chemistry', 1997, 2000)
After deleting tup :
Traceback (most recent call last):
  File "test.py", line 9, in <module>
    print tup;
NameError: name 'tup' is not defined
```

3.1.4. Basic Tuple Operations

Tuples respond to the + and * operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

Python Expression	Results	Description
len((1, 2, 3))	3	Length
(1, 2, 3) + (4, 5, 6)	(1, 2, 3, 4, 5, 6)	Concatenation
('Hi!',) * 4	('Hi!', 'Hi!', 'Hi!', 'Hi!')	Repetition
3 in (1, 2, 3)	True	Membership
for x in (1, 2, 3): print x,	1 2 3	Iteration

3.1.5. Nested Tuples

Tuples can be nested. Tuples can contain other compound objects, including lists, dictionaries, and other tuples. Hence, tuples can be nested inside of other tuples.

Example:

```
tuple1 = (0, 1, 2, 3)
tuple2 = ('pragati', 'college')
```

```
tuple3 = (tuple1, tuple2)
print(tuple3)
```

Output:
((0, 1, 2, 3), ('pragati', 'college'))

3.1.6. Checking the index

The index() method searches an element in a tuple and returns its index.

In simple terms, index() method searches for the given element in a tuple and returns its position.

However, if the same element is present more than once, the first/smallest position is returned.

Remember index in Python starts from 0 and not 1.

The syntax of index() method for is:

```
tuple.index(element)
```

Tuple index() parameters
index() method takes a single argument:

element - element that is to be searched.

Return value from Tuple index()
The index method returns the position/index of the given element in the tuple.

If no element is found, a ValueError exception is raised indicating the element is not found.

Example:

```
vowels = ('a', 'e', 'i', 'o', 'i', 'u')
index = vowels.index('e')
print('The index of e:', index)
index = vowels.index('i')
print('The index of i:', index)
```

Output:
The index of e: 1
The index of i: 2

Reverse Indexing of Tuples in Python

Much similar to regular indexing, here, we use the index inside the square brackets to access the elements, with only one difference, that is, we use the index in a reverse manner. Meaning, the indexing of the elements would start from the last element. Here, we use indexes as −1, −2, −3, and so on, where −1 represents the last element.

Example:

```
tup1= (‘Pragati’, ‘Engineering’, ‘College’)
print (tup1[-1])
```

Output:
College

3.1.7. Counting the elements

The count() method returns the number of occurrences of an element in a tuple.

In simple terms, count() method searches the given element in a tuple and returns how many times the element has occurred in it.

The syntax of count() method is:

```
tuple.count(element)
```

count() Parameters

The count() method takes a single parameter:

element - element whose count is to be found

Return value from count()

count() method returns the number of occurrences of a given element in the tuple.

Example:

```
vowels = ('a', 'e', 'i', 'o', 'i', 'o', 'e', 'i', 'u')
count = vowels.count('i')
print('The count of i is:', count)
count = vowels.count('p')
print('The count of p is:', count)
```

Output:

```
The count of i is: 3
The count of p is: 0
```

3.1.8. List comprehension and Tuples

List comprehensions were added with Python 2.0. Essentially, it is Python's way of implementing a well-known notation for sets as used by mathematicians. In mathematics the square numbers of the natural numbers are, for example, created by { x2 | x ∈ ℕ } or the set of complex integers { (x,y) | x ∈ ℤ ∧ y ∈ ℤ }.

List comprehension is an elegant way to define and create list in Python. These lists have often the qualities of sets, but are not in all cases sets.

List comprehension is a complete substitute for the lambda function as well as the functions map(), filter() and reduce(). For most people the syntax of list comprehension is easier to be grasped.

The following list comprehension creates the Pythagorean triples:

```
>>> [(x,y,z) for x in range(1,30) for y in range(x,30) for z in range(y,30) if x**2 + y**2 ==
z**2]
```

```
[(3, 4, 5), (5, 12, 13), (6, 8, 10), (7, 24, 25), (8, 15, 17), (9, 12, 15), (10, 24, 26), (12, 16, 20), (15, 20, 25), (20, 21, 29)]
>>>
```

Cross product of two sets:

```
>>> colours = [ "red", "green", "yellow", "blue" ]
>>> things = [ "house", "car", "tree" ]
>>> coloured_things = [ (x,y) for x in colours for y in things ]
>>> print coloured_things
[('red', 'house'), ('red', 'car'), ('red', 'tree'), ('green', 'house'), ('green', 'car'), ('green', 'tree'), ('yellow', 'house'), ('yellow', 'car'), ('yellow', 'tree'), ('blue', 'house'), ('blue', 'car'), ('blue', 'tree')]
>>>
```

Generator comprehensions were introduced with Python 2.6. They are simply a generator expression with a parenthesis - round brackets - around it. Otherwise, the syntax and the way of working is like list comprehension, but a generator comprehension returns a generator instead of a list.

```
>>> x = (x **2 for x in range(20))

>>> print(x)

 at 0xb7307aa4>

>>> x = list(x)

>>> print(x)

 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100, 121, 144, 169, 196, 225, 256, 289, 324, 361]
```

3.1.9. Advantages of Tuple over List

Since tuples are quite similar to lists, both of them are used in similar situations. However, there are certain advantages of implementing a tuple over a list. Below listed are some of the main advantages:

- We generally use tuples for heterogeneous (different) data types and lists for homogeneous (similar) data types.
- Since tuples are immutable, iterating through a tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as a key for a dictionary. With lists, this is not possible.
- If you have data that doesn't change, implementing it as tuple will guarantee that it remains write-protected.

3.2. Dictionaries

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

3.2.1. Creating a Dictionary

A Dictionary can be created by placing sequence of elements within curly **{}** braces, separated by 'comma'. Dictionary holds a pair of values, one being the Key and the other corresponding pair element being its **key:value**. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be immutable.

Dictionary keys are case sensitive, same name but different cases of Key will be treated distinctly.

Example:

```
# Creating a Dictionary
# with Integer Keys
Dict = {1: 'Pragati', 2: 'Engineering', 3: 'College'}
print("\nDictionary with the use of Integer Keys: ")
print(Dict)

# Creating a Dictionary
# with Mixed keys
Dict = {'Name': 'PEC', 1: [1, 2, 3, 4]}
print("\nDictionary with the use of Mixed Keys: ")
print(Dict)
```

Output:

```
Dictionary with the use of Integer Keys:
{1: 'Pragati', 2: 'Engineering', 3: 'College'}

Dictionary with the use of Mixed Keys:
{1: [1, 2, 3, 4], 'Name': 'PEC'}
```

Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing to curly braces{}.

Example:

```
# Creating an empty Dictionary
Dict = {}
print("Empty Dictionary: ")
print(Dict)
```

```
# Creating a Dictionary
# with dict() method
Dict = dict({1: 'Pragati', 2: 'Engineering', 3:'College'})
print("\nDictionary with the use of dict(): ")
print(Dict)

# Creating a Dictionary
# with each item as a Pair
Dict = dict([(1, 'Pragati'), (2, 'College')])
print("\nDictionary with each item as a pair: ")
print(Dict)
```

Output:
Empty Dictionary:
{}

Dictionary with the use of dict():
{1: 'Geeks', 2: 'For', 3: 'Geeks'}

Dictionary with each item as a pair:
{1: 'Geeks', 2: 'For'}

3.2.2. Accessing Values

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

Example:

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print ("Name:", dict['Name'])
print ("Age:", dict['Age'])
```

Output:
Name: Zara
Age: 7

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error.

Example:

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print ("Name:", dict['Alice'])
```

```
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    print ("Name:", dict['Alice'])
KeyError: 'Alice'
```

There is also a method called get() that will also help in accessing the element from a dictionary.

Example:
```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print(dict.get('Class'))
```

Output:
First

3.2.3. Adding and modifying an item in a Dictionary

In Python Dictionary, Addition of elements can be done in multiple ways. One value at a time can be added to a Dictionary by defining value along with the key e.g. Dict[Key] = 'Value'. Updating an existing value in a Dictionary can be done by using the built-in update() method. Nested key values can also be added to an existing Dictionary.

Example:
```
# Creating an empty Dictionary
dict = {}
print("Empty Dictionary: ")
print(dict)

# Adding elements one at a time
dict[0] = 'Pragati'
dict[2] = 'Engineering'
dict[3] = 'College'
print("\nDictionary after adding 3 elements: ")
print(dict)

# Adding set of values
# to a single Key
dict['Value_set'] = 2, 3, 4
print("\nDictionary after adding 3 elements: ")
print(dict)

# Updating existing Key's Value
dict[2] = 'Welcome'
print("\nUpdated key value: ")
print(dict)

# Adding Nested Key value to Dictionary
dict[5] = {'Nested' :{'1' : 'Good', '2' : 'Day'}}
print("\nAdding a Nested Key: ")
print(dict)
```

Output:
```
Empty Dictionary:
{}
```

Dictionary after adding 3 elements:
{0: 'Pragati', 2: 'Engineering', 3: 'College'}

Dictionary after adding 3 elements:
{0: 'Pragati', 2: 'Engineering', 3: 'College', 'Value_set': (2, 3, 4)}

Updated key value:
{0: 'Pragati', 2: 'Welcome', 3: 'College', 'Value_set': (2, 3, 4)}

Adding a Nested Key:
{0: 'Pragati', 2: 'Welcome', 3: 'College', 'Value_set': (2, 3, 4), 5: {'Nested': {'1': 'Good', '2': 'Day'}}}

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry.

Example:
```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
dict['Age'] = 8; # update existing entry
dict['School'] = "DPS School"; # Add new entry

print ("Age:", dict['Age'])
print ("School:", dict['School'])
print ("The dictionary is:")
print (dict)
```

Output:
Age: 8
School: DPS School
The dictionary is:
{'Name': 'Zara', 'Age': 8, 'Class': 'First', 'School': 'DPS School'}

3.2.4. Deleting Items

del keyword can be used to in place delete the key that is present in the dictionary. One drawback that can be thought of using this is that is raises an exception if the key is not found and hence non-existence of key has to be handled.

Example:
```
# Python code to demonstrate
# removal of dict. pair
# using del

# Initializing dictionary
test_dict = {"Arushi" : 22, "Anuradha" : 21, "Mani" : 21, "Haritha" : 21}

# Printing dictionary before removal
print ("The dictionary before performing remove is : " + str(test_dict))
```

```
# Using del to remove a dict
# removes Mani
del test_dict['Mani']

# Printing dictionary after removal
print ("The dictionary after remove is : " + str(test_dict))

# Using del to remove a dict
# raises exception
del test_dict['Manjeet']
```

Ouput:
The dictionary before performing remove is : {'Anuradha': 21, 'Haritha': 21, 'Arushi': 22, 'Mani': 21}

The dictionary after remove is : {'Anuradha': 21, 'Haritha': 21, 'Arushi': 22}

Exception :

Traceback (most recent call last):

File "/home/44db951e7011423359af4861d475458a.py", line 20, in

del test_dict['Manjeet']

KeyError: 'Manjeet'

pop() can be used to delete a key and its value inplace. Advantage over using del is that it provides the mechanism to print desired value if tried to remove a non-existing dict. pair. Second, it also returns the value of key that is being removed in addition to performing a simple delete operation.

Example:

```
# Python code to demonstrate
# removal of dict. pair
# using pop()

# Initializing dictionary
test_dict = {"Arushi" : 22, "Anuradha" : 21, "Mani" : 21, "Haritha" : 21}

# Printing dictionary before removal
print ("The dictionary before performing remove is : " + str(test_dict))

# Using pop() to remove a dict. pair
# removes Mani
removed_value = test_dict.pop('Mani')

# Printing dictionary after removal
print ("The dictionary after remove is : " + str(test_dict))
```

```
print ("The removed key's value is : " + str(removed_value))

print ('\r')

# Using pop() to remove a dict. pair
# doesn't raise exception
# assigns 'No Key found' to removed_value
removed_value = test_dict.pop('Manjeet', 'No Key found')

# Printing dictionary after removal
print ("The dictionary after remove is : " + str(test_dict))
print ("The removed key's value is : " + str(removed_value))
```

Output:

```
The dictionary before performing remove is : {'Arushi': 22, 'Anuradha': 21, 'Mani': 21, 'Haritha': 21}
The dictionary after remove is : {'Arushi': 22, 'Anuradha': 21, 'Haritha': 21}
The removed key's value is : 21

The dictionary after remove is : {'Arushi': 22, 'Anuradha': 21, 'Haritha': 21}
The removed key's value is : No Key found
```

3.2.5. Nested Dictionaries

A nested dictionary is a dictionary inside a dictionary. It's a collection of dictionaries into one single dictionary.

```
nested_dict = { 'dictA': {'key_1': 'value_1'},
                'dictB': {'key_2': 'value_2'}}
```

Here, the nested_dict is a nested dictionary with the dictionary dictA and dictB. They are two dictionary each having own key and value.

Creating a Nested Dictionary:

Example:

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}

print(people)
```

Output:

```
{1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}
```

In the above program, people is a nested dictionary. The internal dictionary 1 and 2 is assigned to people. Here, both the dictionary have key name, age , sex with different values. Now, we print the result of people.

Accessing elements of a Nested Dictionary:

To access element of a nested dictionary, we use indexing [] syntax in Python.

Example:

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}

print(people[1]['name'])
print(people[1]['age'])
print(people[1]['sex'])
```

Output:

```
John
27
Male
```

Adding element to a Nested Dictionary

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}

people[3] = {}

people[3]['name'] = 'Luna'
people[3]['age'] = '24'
people[3]['sex'] = 'Female'
people[3]['married'] = 'No'

print(people[3])
```

Output:

```
{'name': 'Luna', 'age': '24', 'sex': 'Female', 'married': 'No'}
```

In the above program, we create an empty dictionary 3 inside the dictionary people. Then, we add the key:value pair i.e people[3]['Name'] = 'Luna' inside the dictionary 3. Similarly, we do this for key age, sex and married one by one. When we print the people[3], we get key:value pairs of dictionary 3.

Adding another dictionary to the nested dictionary:

```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
          2: {'name': 'Marie', 'age': '22', 'sex': 'Female'},
          3: {'name': 'Luna', 'age': '24', 'sex': 'Female', 'married': 'No'}}

people[4] = {'name': 'Peter', 'age': '29', 'sex': 'Male', 'married': 'Yes'}
print(people[4])
```

Output:

```
{'name': 'Peter', 'age': '29', 'sex': 'Male', 'married': 'Yes'}
```

In the above program, we assign a dictionary literal to people[4]. The literal have keys name, age and sex with respective values. Then we print the people[4], to see that the dictionary 4 is added in nested dictionary people.

Deleting elements from a Nested Dictionary:
In Python, we use " del " statement to delete elements from nested dictionary.

Example:
```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
        2: {'name': 'Marie', 'age': '22', 'sex': 'Female'},
        3: {'name': 'Luna', 'age': '24', 'sex': 'Female', 'married': 'No'},
        4: {'name': 'Peter', 'age': '29', 'sex': 'Male', 'married': 'Yes'}}

del people[3]['married']
del people[4]['married']

print(people[3])
print(people[4])
```

Output:
```
{'name': 'Luna', 'age': '24', 'sex': 'Female'}
{'name': 'Peter', 'age': '29', 'sex': 'Male'}
```

In the above program, we delete the key:value pairs of married from internal dictionary 3 and 4. Then, we print the people[3] and people[4] to confirm changes.

How to delete dictionary from a nested dictionary?
```
people = {1: {'name': 'John', 'age': '27', 'sex': 'Male'},
        2: {'name': 'Marie', 'age': '22', 'sex': 'Female'},
        3: {'name': 'Luna', 'age': '24', 'sex': 'Female'},
        4: {'name': 'Peter', 'age': '29', 'sex': 'Male'}}

del people[3], people[4]
print(people)
```

Output:
```
{1: {'name': 'John', 'age': '27', 'sex': 'Male'}, 2: {'name': 'Marie', 'age': '22', 'sex': 'Female'}}
```

In the above program, we delete both the internal dictionary 3 and 4 using del from the nested dictionary people. Then, we print the nested dictionary people to confirm changes.

3.2.7. Difference between a List and a Dictionary

LIST	DICTIONARY
List is a collection of index values pairs as that of array in C++.	Dictionary is a hashed structure of key and value pairs.
List is created by placing elements in [ ] separated by commas ", "	Dictionary is created by placing elements in { } as "key":"value", each key value pair is separated by commas ", "
The indices of list are integers starting from 0.	The keys of dictionary can be of any data type.

The elements are accessed via indices.	The elements are accessed via key-values.
The order of the elements entered is maintained.	There is no guarantee for maintaining order.